ERDC TR-13-8

US Army Corps
of Engineers®
Engineer Research and
Development Center

ERDC
INNOVATIVE SOLUTIONS
for a safer, better world

Center Directed Research Program

Development of Parallel GSSHA

Paul R. Eller, Jing-Ru C. Cheng, Aaron R. Byrd,
Charles W. Downer, and Nawa Pradhan

September 2013

Engineer Research and Development Center

The US Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdc.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at http://acwc.sdp.sirsi.net/client/default.

ERDC TR-13-8
September 2013

Development of Parallel GSSHA

Paul R. Eller and Jing-Ru C. Cheng

Information Technology Laboratory
US Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Aaron R. Byrd, Charles W. Downer, and Nawa Pradhan

Coastal and Hydraulics Laboratory
US Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Final report

Prepared for US Army Corps of Engineers
Washington, DC 20314-1000

Monitored by US Army Engineer Research and Development Center
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

Abstract

GSSHA (Gridded Surface Subsurface Hydrologic Analysis) is a physics-based, distributed, hydrologic, sediment, and constituent fate and transport model. GSSHA can simulate 2D overland flow, 1D stream flow, 1D infiltration, 2D groundwater, and full coupling between groundwater, shallow soils, streams, and overland flow. GSSHA simulations can be very large and time consuming, simulating millions of grid cells over a period of years. In order to run these simulations in a reasonable amount of time, GSSHA must be parallelized to allow many processor systems to effectively run a single GSSHA simulation. Parallelizing GSSHA will enable shorter turnaround time, as well as the study of larger, more complex problems.

This work attempts to fully parallelize GSSHA using MPI, with the ultimate goal of being able to efficiently run GSSHA on thousands of processor cores. We hope to maintain the previous GSSHA functionality by allowing users to run GSSHA without MPI and/or without PETSc. At this point in time we have parallelized and tested the GSSHA code for the overland routing, infiltration, groundwater, soil erosion, channel routing, and lakes processes as well as many other routines needed to run these simulations correctly. We have also parallelized the secant Levenberg-Marquardt alternate run mode. For overland routing we have parallelized the ADE, ADE-PC, and explicit methods. For infiltration we have parallelized the Green and Ampt, Green and Ampt with Redistribution, Multi-layer Green and Ampt, and Richards methods.

Contents

Figures and Tables

Figure

Tables

Preface

This study was conducted for the ERDC Center Directed Research program under the project "Integrated Modeling and Risk Analysis for the Environmental Consequences of Climate Change." The work was performed by the Scientific Computing Research Center (CEERD-IH), US Army Engineer Research and Development Center (ERDC), Information Technology Laboratory (ITL), and the Flood and Storm Protection Division (CEERD-HF), Coastal and Hydraulics Laboratory (CHL).

At the time of publication, Dr. Robert S. Maier was Chief, CEERD-IH; and David R. Richards, was the Technical Director for ERDC-ITL. The Deputy Director of ERDC-ITL was Patti S. Duett and the Director was Reed L. Mosher. Dr. Aaron Byrd was Branch Chief, CEERD-HFH; and Dr. Ty V. Wamsley was the Division Chief, CEERD-HF, William R. Curtis, CEERD-HT, was the Technical Director for ERDC-CHL. The Acting Deputy Director of ERDC-CHL was Richard Styles and the Acting Director was Jose E. Sanchez.

COL Jeffrey R. Eckstein was the Commander of ERDC, and Dr. Jeffery P. Holland was the Director.

1 Introduction

GSSHA (Gridded Surface Subsurface Hydrologic Analysis) is a physics-based, distributed, hydrologic, sediment, and constituent fate and transport model. GSSHA can simulate 2D overland flow, 1D stream flow, 1D infiltration, 2D groundwater, and full coupling between groundwater, shallow soils, streams, and overland flow. This can allow GSSHA to simulate a wide variety of environments to analyze future conditions, management scenarios, flood control, sediment transport, and pollutant transport as well as many other types of problems.

GSSHA simulations can be very large and time consuming, simulating millions of grid cells over a period of years. In order to run these simulations in a reasonable amount of time, GSSHA must be parallelized to allow many processor systems to effectively run a single GSSHA simulation. Individual workstations often contain multi-core processors, while larger clusters or supercomputers can provide users with access to hundreds of thousands of processors. Parallelizing GSSHA will allow users to obtain results in a shorter amount of time, as well as study larger, more complex problems.

GSSHA has previously been given some parallel functionality through parallelizing key components of the code using MPI or OpenMP. This work attempts to fully parallelize GSSHA using MPI, with the ultimate goal of being able to efficiently run GSSHA on thousands of processor cores. We hope to maintain the previous GSSHA functionality by allowing users to run GSSHA without MPI and/or without PETSc. At this point in time we have parallelized and tested the GSSHA code for the overland routing, infiltration, groundwater, soil erosion, channel routing, and lakes processes as well as many other routines needed to run these simulations correctly. We have also parallelized the secant Levenberg-Marquardt alternate run mode. For overland routing we have parallelized the ADE, ADE-PC, and explicit methods. For infiltration we have parallelized the Green and Ampt, Green and Ampt with Redistribution, Multi-layer Green and Ampt, and Richard's methods.

Background

Multiple software packages are used to efficiently parallelize GSSHA. MPI, DBuilder, and PETSc are used to develop the parallel version of GSSHA.

MPI

MPI (2012) (Message Passing Interface) is standardized and portable message passing system for parallelizing code on a wide variety of systems, in particular large distributed systems. MPI runs on supercomputers as well as desktop computers, allowing all GSSHA users to take advantage of the MPI parallelism provided they have MPI installed on their system. GSSHA is primarily parallelized with MPI or other software packages that use MPI.

DBuilder

The ERDC Information Technology Laboratory has developed a software package called DBuilder (Hunter and Cheng 2005; Campbell et al. 2010), which provides support for domain partitioning, parallel data management, and coupling coordination. DBuilder provides users with a simplified interface to MPI based parallelization routines. Taking advantage of DBuilder allows us to simplify the parallelization process. The domain partitioning code allows us to partition the 2D GSSHA domain, while the coupler code allows us to pass data between the partitioned 2D domain and the smaller 1D domain stored on each processor.

PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) (Balay et al. 1997; Balay et al. 2008; Balay 2012) provides users with access to a suite of data structures and routines for parallel scientific applications. These routines include a wide variety of fast, scalable linear solvers and preconditioners. The original GSSHA groundwater solver is a line successive over-relaxation solver, which is hard to parallelize well. The GSSHA groundwater routine is modified to use PETSc to solve the linear systems. Interfacing GSSHA with PETSc gives users access to many different state-of-the-art linear solvers and preconditioners that can be used to more quickly and more accurately solve linear systems.

General Parallelization

GSSHA simulates a wide variety of processes, including both 2D and 1D domains. The 2D domains use a fixed grid where the grid cells can be divided among the processors, allowing each processor to operate on a subset of the overall 2D domain. The 1D domain is much smaller in comparison and does not parallelize as well as the 2D domain. Therefore the full 1D domain is simulated on each processor. A coupler is used to transfer data between the 1D and 2D domains.

GSSHA uses 1-based indices, while DBuilder and most MPI routines use 0-based indices. The MPI GSSHA code is written in mpi *.cpp files. Macros are written to properly pass the 1-based arrays to functions that require 0-based arrays, as well as to simplify the GSSHA code. Some MPI routines require careful changes to produce the correct results when using both 1-based and 0-based arrays in the same routine.

Some GSSHA routines required significant changes to produce correct results efficiently on multiple processors. Some routines would loop over the 2D domain and modify neighboring grid cells instead of just modifying the current grid cell. Since the neighboring grid cells are sometimes stored on another processor, this required some routines to be rewritten to only modify the current grid cell. Other routines were written in ways that limit parallelism, such as using a LSOR (Line Successive Over Relaxation) linear solver rou tine that operates on each row of the 2D grid one at a time.

These routines needed to be replaced, such as by replacing the LSOR routine with the PETSc linear solver.

Most functions related to the MPI routines are called DBuild * or mpi *. These routines have #ifdef preprocessor directives around them to allow the code to be compiled with or without MPI. We use the GMPI flag for the MPI sections of code. Other parts of GSSHA are rewritten to work correctly when compiled serially or with MPI. While the parallelized GSSHA code is primarily tested and run on linux systems, the code is also designed to work correctly and tested on Windows systems.

A primary data structure is created and added to the GSSHA main var struct containing all of the needed variables. This allows MPI DATA to be passed to each function that is parallelized or that calls a parallelized function. Define statements are used to create an empty MPI DATA

structure for non-MPI simulations. This data structure is primarily used in the MPI routines. Macros are developed to simplify GSSHA's interaction with the MPI code. In most cases, the MPI DATA struct will be passed into many GSSHA routines and then the MPI routines will modify the variables inside this struct.

```
typedef struct MPI_DATA {

     // 2d domain struct
     DB_Subdomain vtxDomain2d; // 2-D data structure

     // DBuilder types
     DB_Type intType;          // Integer data type
     DB_Type doubleType;       // Double data type
     DB_Type sedType3d;        // 3d sediment type

     // MPI types
     MPI_Datatype sedMpiType3d;// 3d sediment MPI type

     // Mesh information
     int ownedElements;        // Number of owned elements
     int ghostElements;        // Number of ghost elements
     int localElements;        // Total number of local
                               // elements (owned + ghost)
     int globalElements;       // Number of global elements

     // 1d domain struct
     DB_Subdomain vtxDomain1d; // 1-D data structure

     // 1-D and 2-D Coupler
     DB_Coupler vtxCoupler;    // DBuilder coupler between
                               // 1-D and 2-D domains
     // Alt Run Modes
     MPI_Comm GSSHA_COMM;      // MPI Communicator
     int mpi_group;            // MPI group number

     // MPI I/O
     int mpi_binary_io;            // binary I/O flag
     MPI_Datatype mpi_file_double; // MPI double file
     MPI_Datatype mpi_file_int;    // MPI int file

     // PETSc structs
     PETSC_DATA *petsc_data;   // PETSc solver struct

} MPI_DATA;
```

The primary MPI data structure also contains a PETSC DATA data structure for the petsc solver. This stores information related to the linear systems that must be solved. Since the structure of the mesh is constant throughout the simulation, we can create a PETSc matrix, vectors, and KSP solver at the beginning of the simulation, and then set double values of the matrix and vectors in the PETSc solver code.

```
typedef struct PETSC_DATA {

     int prev_global_rows;      // Number of rows of data for
                                // prior processors
     int *petsc_map;            // Mapping from global number
                                // to local number
     int *dia, *oia, *dja, *oja;   // Matrix structure arrays
                                    // for split matrix
     double *dval, *oval;       // Matrix value arrays for
                                // split matrix
     Mat A;                     // Matrix for linear solver
     KSP ksp;                   // Linear solver data structure
     Vec x, b;                  // Solution and rhs vectors for
                                // linear solver
     int maxdj, maxoj;          // Max number of elements in a
                                // diagonal and off-diagonal
} PETSC_DATA;
```

2 2D Domain Decomposition

The 2D domain is partitioned using DBuilder, which calls ParMETIS (Karypis 2012; Karypis and Schloegel 2011) to partition the 2D domain. An equal portion of the 2D mask containing the 2D map of which grid cells are being used during the simulation is read on each processor. By reading an equal number of rows on each processor instead of the entire mask, we reduce the memory requirements and reduce the amount of time spent in I/O. This mask is used to create a list of neighbors for each grid cell, which is used by DBuilder to partition the grid. We also compute the values for the abovecon, leftcon, rightcon, belowcon, maskrow, and maskcol arrays using this 2D mask and add them to a user information array. The DBuilder partition routine then passes the data for the user information array to the processor that locally owns the data after the partition.

DBuilder returns a data structure containing the partitioned domain, including a list of all grid cells owned by each processor. We also store a layer of ghost cells bordering the 2D grid on each processor. This allows us to store a copy of data stored on other processors that is needed to complete computations for data stored on the current processor. We can complete computations that require data from neighboring grid cells while minimizing the time spent passing data between processors.

Once the domain has been partitioned, we delete the 2D mask array in order to greatly reduce the amount of memory used by GSSHA in comparison to storing the full 2D mask array on each processor. The mask array is replaced with partioned maskrow and maskcol arrays that contain the global row and column for each grid cell.

With the addition of the partitioned domain, we create the two new variables ONUM and LNUM in addition to GNUM to provide information about the number of grid cells used during the simulation. ONUM stores the number of owned grid cells and LNUM the sum of the owned grid cells and the ghost grid cells. ONUM replaces GNUM throughout GSSHA for loops over the entire 2D domain and for allocating arrays in some cases where the array is not involved with computations involving neighboring grid cells. LNUM replaces GNUM throughout GSSHA for allocating arrays

that are involved with computations involving neighboring grid cells and for some loops where each processor has the necessary information to modify the ghost cells without transferring data between each processor.

Basic Examples

Once we have created the partitioned domain, we can modify the GSSHA code to properly interact with the partitioned grid. There are a number of common communication patterns that are used throughout GSSHA that we will discuss below.

We modify partitioned arrays by looping over the locally owned grid cells. If the array we are modifying is involved in computations using the current grid cell and a neighboring grid cell, then we must call a global update to make sure that the ghost values are accurate.

```
for(i=1; i_ONUM; i++)
   array1[i] = coef*data[i];
DBuild_Global_dupdate(array1,mpi_data);

for(i=1; i_ONUM; i++)
   array2[i]=coef*(array1[i+1]-array1[i]);
DBuild_Global_dupdate(array2,mpi_data);
```

Throughout GSSHA there are some values that are computed across the entire domain. Some of these values are running sums, while others are computed each time-step. Additionally maximum or minimum values are computed across the entire domain in some cases. Each of these situations requires some changes to the code.

```
double sum_value=0.0;
for(i=1; i_ONUM; i++) {
   sum_value += array[i];
}
DBuild_Sum_Double(sum_value,mpi_data);
main_value += sum_value;
```

Error checking code must be changed to properly respond to errors when using multiple processors, especially when error checking inside loops over the 2D domain. The error message must be broadcast to all processors after the loop is completed so that all processors are aware that an error has occurred and can all have the same response. In some cases, instead of computing an error that causes the code to exit, an error tolerance will be computed that causes the time-step to be reduced or another action to take place to improve the accuracy of the code, requiring similar changes.

```
for(i=1; i_ONUM; i++) {
   if(isnan(array[i])!=0) {
     printf("Error message");
     errptr.flag=4000;
     break;
   }
}
DBuild_Max_Int(errptr.flag);
if(errptr.flag==4000) {
  main_err(&errptr);
  return;
}
```

Domain Coupling

We create a coupler between the 2D domain and the streamcells for the 1D domain. The streamcells are 2D grid cells that contain 1D grid cells. The coupler creates a map from the 2D domain that is partitioned across all processors to a streamcell array that is stored on each processor. This array contains nstreamcells gridcells in indices 1 to nstreamcells; therefore, we do not need to use gst_gnum to determine the grid number and can instead set the grid number using the current loop iteration for the streamcells. For any routine with computations that use both the 1D and 2D domains, we create a streamcell array for any 2D array using the coupler at the beginning of the routine. These streamcell arrays interact with the 1D domain, which is stored on each processor. Once the 1D computations are complete, we then copy any modified streamcell arrays back to the 2D grid.

```
// Get array of streamcells
stream=(double*)malloc(nstreamcells+*sizeof(double));
DBuild_Coupler_dupdate_to_stream(stream,global,
                                 DB_D2TOD1,mpi_data);

// Modify array of streamcells
for(sc=1; sc_nstreamcells; sc++) {
   for(frag=1; frag_ncellsingrid[sc]; frag++) {
#ifdef GMPI
     grid=sc;
#else
     grid=gst_gnum[sc][frag];
#endif
     stream[grid]+=gst_val*1d_val;
     array1d[node][link]=gst_val*val;
   }
}

// Update original 2-D array using streamcells
DBuilder_Coupler_dupdate_to_global(global,
                                 stream,mpi_data);

// Free streamcell array
free(stream);
```

MPI Macros

A number of macros have been added to simplify the interaction between GSSHA and DBuilder. This includes macros for updating ghost cells, converting between local and global indices, coupling the 1D and 2D domains, and computing a reduction on a variable stored on all processors. These arrays take into account that the GSSHA arrays are stored using 1-based indexing and the DBuilder arrays are stored using 0-based indexing.

Updating ghost cells

DBuild_ Global dupdate/iupdate(array,mpi_data)
Given an array partitioned across all processors, updates the ghost gridcells with values stored on other processors.

DBuild_ Global dupdate 3d(array,type,mpi_data)
Given a partitioned 3d array and DBuilder type, updates the ghost gridcells with values stored on other processors.

Converting between local and global indices

DBuild_Get_local_fnumber(global,local,mpi_data)
Given the global gridcell index, returns the local gridcell index using 1-based indexing, returning -1 if the global index is not stored locally.

DBuild_Get_global_fnumber(local,global,mpi_data)
Given the local gridcell index, returns the global gridcell index using 1-based indexing.

Coupling 1D and 2D domains

DBuild_Coupler_dupdate/iupdate_to_stream(array1d, array2d, mpi_data)
Updates the 1D stream array with values from the 2D global array.

DBuild_Coupler_dupdate/iupdate_to_global(array2d, array1d, mpi_data)
Updates the 2D global array with the values from the 1D stream array.

DBuild_Coupler_dupdate_3d_to_stream(array1d, array2d, mpi_data)

Updates the three dimensional 1D stream array with values from the three-dimensional 2D global array.

DBuild_Coupler_dupdate_3d_to_global(array2d, array1d, mpi_data)
Updates the three- dimensional 2D global array with the values from the three-dimensional 1D stream array.

MPI Allreduce across all processors

DBuild_Sum_Double/Int(value,mpi_data)
Computes the sum of the value across all processors.

DBuild_Max_Double/Int(value,mpi_data)
Computes the maximum value of a variable across all processors.

3D Arrays

In order to work with parallelized 3d arrays, we developed the DARRAY 3D struct found in array 3d.h and array 3d.cpp. This struct allows us to more easily create and manipulate 3D arrays using MPI. The DARRAY 3D struct stores data in a 1D array, and contains variables to keep track of the size of the array (number of locally owned grid cells), the number of rows per grid cell, and the number of columns per grid cell. Accessor methods are provided to allow users to modify the values of 3D arrays. This routine was developed for the soil erosion routine. Below is the struct definition. There are examples of code using the 3D arrays in the soil erosion routine discussion below.

```
1 typedef struct DARRAY_3D {
2      DARRAY_3D();
3      DARRAY_3D(int size, int rows, int cols);
4      ¬DARRAY_3D();
5      double& operator()(int i, int j, int k);
6      double& index(int i, int j, int k);
7
8      // Local array information
9      int size;
10     int rows, cols;
11     double *data;
12
13 } DARRAY_3D;
```

3 GSSHA Routine Parallelization

Next we discuss some of the key routines that have been parallelized. In addition to these key routines, a large number of smaller routines have been parallelized throughout the GSSHA code. Most of these routines use parallelization techniques similar to the ones discussed throughout this report.

Overland Flow Routing

The overland flow routing ADE and ADE-PC route overland ADEPC routines and explicit method route_overland_adtv_wetlands routine are all parallelized. The ADE and ADE-PC methods required some minor modifications to be parallelized, while the explicit method required some more significant modifications.

The ADE and ADE-PC methods modified the FLOWX, FLOWY, and update_h routines to compute only the locally owned grid cells and then update the ghost nodes after each loop. The code to compute the outcoordinate grid cell was modified to verify that the grid cell is on the current processor. MPI Allreduce routines are called to compute the sum or maximum value of certain variables across all processors. Below is an example of changes that were made.

```
1 double route_overland_ADEPC(...) {
2   while(time>1e-16) {
3      FLOWY(h,qy,...);
4      update_h(h,qy,...);
5      ...
6   }
7 }
8
9 void FLOWY(h,qy,...) {
10  for(i=1; i_ONUM; i++)
11     qy[i]=qy[i]+val*(h[i+1]-h[i]);
12  DBuild_Global_dupdate(qy,mpi_data);
13 }
14
15 void update_h(h,qy,...) {
16  for(i=1; i_ONUM; i++) {
17     h[i]=h[i]+val(qy[i+1]-qy[i]);
18     if(h[i]<val2)
19        loopbreak=2;
20  }
```

```
21  DBuild_Max_Int(loopbreak);
22  DBuild_Global_dupdate(h,mpi_data);
23 }
```

The explicit method originally computes the *dh* values for the current gridcell and neighboring grid cell in the loops computing *qx* and *qy*. There is then a loop to use *dh* to update h. This is modified to compute only *qx* and *qy* in separate loops, and then to compute the new h values based on *qx* and *qy*, similar to the ADE and ADE-PC methods. Below is an example of the changes.

```
1 // Compute qx
2 for(j=1; j_ONUM; j++)
3   qx[j]=qx[j]+newvalue;
4 DBuild_Global_dupdate(qx,mpi_data);
5
6 // Compute qy
7 ...
8
9 // Compute h
10 for(j=1; j_ONUM; j++) {
11  if(leftcon[j]!=0)
12     qx2l = qx[leftcon[j]];
13  else
14     qx2l = 0.0;
15  if(abovecon[j]!=0)
16     qy2a = qy[abovecon[j]];
17  else
18     qy2a = 0.0;
19  h[j] += (qx2l-qx[j]) + (qy2a-qy[j]);
20 }
21 DBuild Global dupdate(h,mpi data);
```

Infiltration

The Green and Ampt inf gna routine, Green and Ampt with Redistribution inf_redist routine, Multi-layer Green and Ampt inf_gna_multi routine, and Richards equation richards_solver routine are parallelized. Each routine required modifications to only compute the locally owned grid cells and then update the h array at the end of the methods, similar to the modifications for the overland flow routing routines.

Groundwater

The groundwater routine is modified to use a PETSc linear solver instead of the original LSOR linear solver for parallel GSSHA runs. The original LSOR solver is still used for serial runs. We also allow users to call the original LSOR solver while using MPI. However, using this routine with

MPI requires passing all data to processor 0, which solves the linear system and passes the results to the other processors. This results in much slower running times, so we only recommend using this for purposes such as testing the accuracy of the code in comparison to the serial version.

The groundwater routine is primarily modified to compute the locally owned grid cells for a number of arrays and then update the ghost cells. Many sections of code are modified to compute the sum or maximum value of an array using MPI_Allreduce routines. The major changes are related to the introduction of the PETSc linear solver in the mpi_init_petsc_solver (called by the initialization section of the main_gssha_function) and mpi_petsc_solver (called by the groundwater function) routines.

PETSc Linear Solver

The LSOR linear solver computes the new values for each row or column of the 2D grid by generating and solving a tridiagonal matrix for each row or column, using the newly computed values from the previous row or column. This greatly limits parallelization due to only solving a single row or column at a time and containing a significant amount of code that is serial in nature.

Instead, we can generate a linear system using any 2D partition and use PETSc to solve this linear system. We create a PETSc CSR matrix with split arrays, allowing us to create a matrix structure once during the initialization phase, and then to set the values of the matrix each time we solve the linear system. Each processor generates ONUM rows of the matrix. The split array structure requires that we create a CSR diagonal block array and a CSR off-diagonal block array. The diagonal block array holds the data for the grid cells stored on the current processor, while the off-diagonal block holds the data for the grid cells stored on other processors. Once we have created a matrix and set the values of the matrix, we simply call the PETSc solver, which returns a solution vector.

Adding PETSc functionality allows us to use any linear solvers and preconditioners provided by PETSc that are compatible with the split array matrix format, as well as any new linear solvers that are created in the future.

Soil Erosion

The soil-erode routine required many minor changes related to modifying loops to only compute the values for grid cells stored on the locally owned grid cells and MPI Allreduce calls to sum or find the maximum value of a variable. However some major changes were needed to correctly parallelize the section of code for computing the sediment mass balance. The original routine was a single large loop computing the values for grid cells in the *x*-direction and then the *y*-direction. Each iteration would modify values of the current grid cell and two neighboring grid cells. This results in problems when dividing the domain among many processors, as some computations modify values stored on other processors.

To correctly parallelize this, we needed to divide the single loop into four separate loops, allowing us to modify only the values stored on the current processor, and to update the ghost nodes after computing values in the *x*- and *y*-directions. We first compute temporary values in order to ensure that all computations are not affected by previous iterations of the loop. We then update the values of the main arrays using these temporary arrays. We then call a global update to update the ghost nodes of the main 3D arrays. Since each iteration of each loop modifies both the current grid cell and a neighboring grid cell, we loop over both the locally owned grid cells and the ghost grid cells. This ensures that all necessary computations are completed, although there are extra computations since we modify values in the grid cells which are overwritten when performing a global update.

```
1 // Outline for sediment mass balance computations:
2 Compute temp values for each grid cell in x-direction
3 Compute locally owned grid cells
4 Global update for vol_sed
5 Compute temp values for each grid cell in y-direction
6 Compute locally owned grid cells
7 Global update for vol sed
```

In order to more clearly see how the code is modified and how the DARRAY 3D class is used, below is a simplified code snippet from this routine. The xysedfract and xysed arrays are computed in the previous loop using sed fract and vol sed. This setup prevents the previously computed values in a loop from affecting the values computed later in the loop, ensuring that we obtain the same solution on any number of processors.

```
for(j=1; j_LNUM; j++) {
   for(k=1; k_NUMS; k++) {
      // Determine outgoing and receiving cells
      if(qxx[j]<0.0) {
         outgo=rightcon[j];
         recvg=j;
      } else {
         outgo=j;
         recvg=rightcon[j];
      }

      // Update gridcells with previously computed values
      (*sed_fract)(outgo,k,2) = xysedfract[j][k];
      (*vol_sed)(outgo,k,2) -= xysed[j][k];
      (*vol_sed)(recvg,k,2) += xysed[j][k];

      // Zero out arrays index for next direction
      xysed[j][k]=0.0;
      xysedfract[j][k]=0.0;
   }
}
```

Channel Routing

Parallelizing the channel routing process required modifying a number of routines including ex flow route, lateral inflow, and gw chan exchange. The 1D channel routing domain is also generally fairly small in comparison to the 2D domain, so there are limited opportunities for speedups. Therefore we set up the 1D domain to be computed on each processor. This requires us to copy data from the partitioned 2D domain to every processor at the beginning and end of many of the 1D channel routing routines.

Each of these routines use a similar pattern as shown in the earlier section on coupling the 1D and 2D domains. The 2D array is copied to a streamcell array stored on each processor, which is then used for the 1D computations. If this array is modified during the course of the routine, these streamcell arrays are copied back to the 2D array. This process allows us to limit the amount of the 1D code we must modify, allowing us to more quickly parallelize the code and limit the number of changes we must make.

Lakes

The 1D lake_calcs routine is parallelized similar to the 1D channel routing routine by keeping a full copy of the lake data on each processor. This routine loops over all of the lake cells on every processor and modifies the 2D array grid cells stored on the local processor. Some values computed on each grid cell are broadcast to all processors to ensure that every processor ends up with the same lake values.

SLM Alternate Run Mode

The secant Levenberg-Marquardt local search method is modified to run parallel GSSHA simulations, as well as to run multiple parallel GSSHA simulations at the same time.

The GSSHA code must first be modified to work with linux. This requires converting the postGSSHA.bat script to a postGSSHA.sh script, as well as using slightly modified versions of otl2ssf.exe and tsproc compiled using linux. These modifications allow these routines to work with multiple sets of input and output files at once. When calling multiple instances of the main gssha routine with different input values, only one processor per instance will call postGSSHA.sh.

Next we modify GSSHA so that we can run multiple different GSSHA simulations at once. The main GSSHA routine is modified to accept a MPI Communicator and a group number as input. The GSSHA code is modified to use this communicator when calling MPI functions, and appends the group number to any output files. Therefore if eight GSSHA simulations are run at once, then eight variations of each output file are produced.

The ModelSensitivityMatrix loop to compute the Jacobian is modified to run multiple GSSHA simulations at once based on the number of Jacobian updates needed and the number of available processors. The code attempts to take advantage of all available processors for each iteration of the loop. If there are more Jacobian updates than processors, then there will be multiple sets of Jacobian updates using mostly one processor per run. If there are many more processors than Jacobian updates, then each run will use multiple processors. Each call to main gssha will result in a different set of output files.

At the end of the ModelSensitivityMatrix loop, we update the solution array on all processors. Each processor computes some new values for this array, but at the end of this loop, we want every processor to have the same values for each variable. We loop over the solution array and determine which processor computed the new value for each array index and then broadcast this value to all processors.

When other parts of the SLM code call main gssha to compute initial values, run models for new lambda values, etc, one group and all available processors are used.

4 Parallel I/O

The original GSSHA code primarily read and wrote ascii files. In order to speed up the I/O code and allow the I/O to work correctly for large grids, we need to convert the input files to ascii, develop binary routines capable of quickly reading and partitioning the data, and then convert the output files from binary back to ascii. This requires the developing of pre- and post-processing routines as well as the MPI GFILE class that handles the parallel file I/O using MPI I/O. These I/O routines are added in addition to the original serial ascii routines.

Pre- and Post-processing

A pre-processing program called pre gssha is developed that must be run prior to the main GSSHA simulation. This routine reads the project file and picks out the input files that must be converted from ascii to binary. The GSSHA code to read the project file is adapted to a simpler version in the file pre post readprj.cpp. This code reads in each line from the project file, and extracts and saves the names of any input or output files. The pre-gssha routine then calls a function to convert the file from ascii to binary. This routine first writes any header data to the top of the file. The routine then reads in each line of an input file and each line of the mask file, and then writes the grid cell values from the input file for each non-zero entry in the mask file to the binary file. This routine only needs to be run once after the input files are created or modified.

A similar routine called post-gssha is used for post-processing. This routine also uses the pre post readprj.cpp file to process the project file and extract input and output file names. This routine then calls a function to conver the binary files to ascii files. This routine first converts the binary header information to ascii information, and then reads in each line of the binary output file and each line of the mask file. The routine then writes the zeros to the locations in the 2D grid for a zero entry in the mask file, and writes values from the binary files for each non-zero entry in the mask file. This routine must be run after every GSSHA simulation in order to convert the new output files from binary to ascii.

GSSHA MPI File Class

The MPI_GFILE class is designed to handle opening, reading, writing, and closing MPI files for GSSHA. New I/O routines have been written for the parallel MPI files in addition to the original ascii files.

The MPI_GFILE class contains the name of the original and binary files, the MPI File object, an offset used to track the writing location in the file, and a boolean value to determine if the file is open.

```
typedef class MPI_GFILE {

     char *filename;         // File name
     char *binfilename;      // Binary file name

     MPI_File fptr;          // MPI_File object

     int offset;             // Current read/write location
                             // in MPI_File
     bool opened;            // True if file has been opened,
                             // false if not
} MPI_GFILE;
```

In the initialization code, the routine mpi init file types uses the 2D partition to create an MPI type used to read and write data from files. Separate routines are created for double and integer file types.

```
// Create double mpi file type
MPI_Type_create_indexed_block(mpi_data->ownedElements,1,
     mpi_data->vtxDomain2d.globalNumber,
     MPI_DOUBLE,&mpi_data->mpi_file_double);
MPI_Type_commit(&mpi_data->mpi_file_double);

// Create int mpi file type
MPI_Type_create_indexed_block(mpi_data->ownedElements,1,
     mpi_data->vtxDomain2d.globalNumber,
     MPI_INT,&mpi_data->mpi_file_int);
MPI_Type_commit(&mpi_data->mpi_file_int);
```

Once these MPI types are created, we can read or write data from files by setting a view using this MPI type, and then write the data to the file. Below is an example based on the MPI GFILE code.

```
// Writing header data to a file
MPI_File_set_view(fptr,offset,MPI_DOUBLE,MPI_DOUBLE,
                          "native",MPI_INFO_NULL);
MPI_File_write_all(fptr,&val1,1,MPI_DOUBLE,&status);
MPI_File_write_all(fptr,&val2,1,MPI_DOUBLE,&status);
MPI_File_write_all(fptr,&val3,1,MPI_DOUBLE,&status);
MPI_File_write_all(fptr,&val4,1,MPI_DOUBLE,&status);
```

```
offset+=sizeof(double)*4;

// Writing double array to a file
MPI_File_set_view(fptr,offset,MPI_DOUBLE,
     mpi_data->mpi_file_double,"native",MPI_INFO_NULL);
MPI_File_write_all(fptr,data,mpi_data->ownedElements,
                           MPI_DOUBLE,&status);
```

This allows the GSSHA code to call a few fairly simple functions to read and write data. Below is an example that opens a file, reads the header data to many variables, reads the file data to an array, and then closes the file.

```
MPI_GFILE *mpi_fptr;
mpi_fptr.open_read_file(filename,mpi_data);
mpi_fptr.read_header(&val1,&val2,...,&valn,mpi_data);
mpi_fptr.read_map_double(data,mpi_data);
mpi_fptr.close_file();
```

Separate routines are written for each type of GSSHA file in order to make sure that the MPI version produces the same output as the original routines.

5 Run Procedure

Once a GSSHA project has been developed, there are a number of steps needed to run the MPI version. GSSHA users will first need to compile GSSHA using the compile all.sh script. This will compile the code for GSSHA, DBuilder, and ParMETIS and produce an executable called gssha. This compile script will also compile a program for the secant Levenberg-Marquardt alternate run mode, the pre gssha program, and the post-gssha program.

Once GSSHA is compiled, the pre gssha routine must be run to convert the input files into a binary format for MPI GSSHA. This command only needs to be run once for a set of input files, although if the input files are changed, you may need to rerun pre gssha. Next we run the GSSHA simulation using mpirun (or a similar mpi command) on multiple processors. Finally we run post gssha in order to convert the binary output files into ascii binary files.

```
// In GSSHA code directory /home/user/GSSHA
sh compile_all.sh

// In GSSHA project directory
// /work/user/GSSHA_Projects/BasicGSSHA
./pre_gssha basic_ov.prj
mpirun -np 4 ./gssha basic_ov.prj
./post gssha basic ov.prj
```

For small, quick simulations, GSSHA user may be able to use a desktop system or an interactive session on a HPC (High Performance Computing) system. However for large, time-consuming simulations, GSSHA users will need to submit a job to a batch queue on a HPC system. This will allow users to submit a job that will run when there is space available on the HPC system. The queue system tends to start smaller, shorter runs more quickly, so users should take this into account when selecting the number of processors to use and the wall time for their run.

```
#!/bin/sh
#PBS -l ncpus=256
#PBS -l walltime=12:00:00
#PBS -q standard
#PBS -A ACCOUNT
#PBS -N gssha_256

cd /work/user/GSSHA_Projects/BasicGSSHA

mpirun -n 256 /home/user/GSSHA/gssha_mpi/gssha
   basic ov.prj > gssha 256 output.txt
```

6 Performance Results

In order to demonstrate the performance of GSSHA, we run tests using the large New Orleans and Fort Stewart models. The New Orleans model contains about 4.5 million grid cells. The simulation runs for 1 hour with a 0.01 second time-step size, simulating overland flow using the ADE method. Tests are run on Garnet, a Cray XE6, using from 16 to 4096 processors. We show the computation time and total running time in Table 1, where the computation time is the time spent in the main time-stepping loop without I/O, and total running time includes initialization time and I/O time. These results demonstrate that we can obtain significant speedups using up to about two thousand processors.

Table 1. Run times for a 1 hour New Orleans simulation

Processors	Computation Time	Total Time
16	64638.29	64675.77
512	2526.98	2566.40
1024	1868.22	1913.16
2048	1502.39	1555.89
3072	1818.73	1881.07
4096	2034.20	2108.93

The Fort Stewart model contains about 45 million grid cells. The simulation runs for 48 hours with a 10 second time-step size, simulating overland flow using the ADE method. Tests are run on Garnet, a Cray XE6, using from 16 to 1024 processors. The results in Table 2 demonstrate that we can obtain significant speedups on at least 1024.

Table 2. Run times for a 48 hour Fort Stewart simulation

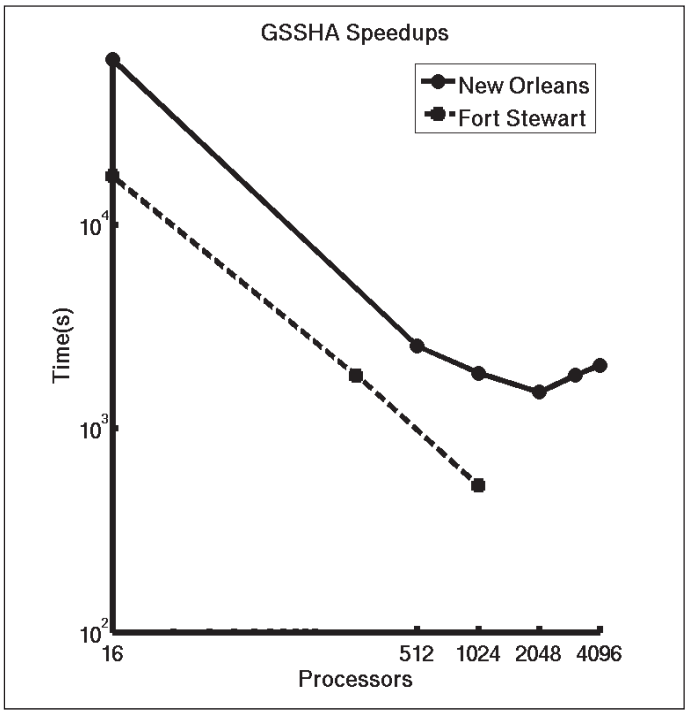
Processors	Computation Time	Total Time
16	17282.72	19519.51
256	1813.82	4026.44
1024	523.35	2763.07

The exponential speedup plot in Figure 1 more clearly shows that we obtain near linear speedups on up to 512 to 1024 processors for these two models. However, the New Orleans model shows that as the overhead increases and amount of work per processor decreases, the running times start to increase on over 2048 processors. Larger datasets and more optimization will be necessary to obtain better speedups.

7 Conclusion

This report discusses the implementation of MPI within GSSHA, using DBuilder to assist with the parallelization. The 2D domain is partitioned across all processors, while the 1D domain is computed on each processor, using a coupler to pass data between the 2D and 1D domains. The original LSOR groundwater linear solver is replaced with the PETSc linear solvers, providing access to a wide variety of state-of-the-art parallel linear solvers. The overland routing, infiltration, ground-water, soil erosion, channel routing, and lakes processes and the secant Levenberg-Marquardt alternate run mode are parallelized. Results demonstrate that we are capable of obtaining significant speedups on up to about two thousand processors, with potential for speedups on larger numbers of processors for larger problems.

Figure 1. Exponential speedup plot for New Orleans and Fort Stewart Simulations.



A significant amount of future work still remains in order to finish parallelizing GSSHA. A number of routines and input options still need to be parallelized, such as routines related to wetlands, snow, and many of the alternate run modes. More detailed testing will be needed to verify that

all input options and original GSSHA functionality work correctly in the parallel version. There are many opportunities to optimize the code for better performance, especially with the 1D routines, which currently run a copy of the 1D model on every processor. Many of the 2D routines also have opportunities for improvement since the primary focus up to this point has been to ensure that the code runs accurately in parallel. Further maintenence will also be needed to ensure that MPI GSSHA continues to produce accurate solutions as more new routines and code are introduced to GSSHA.

References

Message Passing Interface (MPI) Forum. 2012. forum home page, 2012. http://www.mcs.anl.gov/research/projects/mpi/.

Hunter, R. M. and J-R. C. Cheng. 2005. *Dbuilder: A parallel data management toolkit for scientific applications*. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, volume 2.

Campbell, T., R. Allard, R. Preller, L. Smedstad, A. Wallcraft, Sue Chen, Hao Jin, S. Gabersandek, R. Hodur, J. Reich, C.D. Fry, V. Eccles, Hwai-Ping Cheng, J.-R.C. Cheng, R. Hunter, C. DeLuca, and G. Theurich. 2010. Integrated modeling of the battlespace environment. *Computing in Science Engineering*, 12(5):36–45.

Balay, S., K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. 2012. PorTable, extensible toolkit for scientific computation, 2012. http://www.mcs.anl.gov/petsc.

Balay, S., K. Buschelman, V. Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. 2008. *PETSc users manual*. Technical Report ANL-95/11 – Revision 3.0.0, Argonne National Laboratory, 2008.

Balay, S., W. D. Gropp, L. C. McInnes, and B. F. Smith. 1997. *Efficient management of parallelism in object oriented numerical software li- braries*. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, Modern Soft- ware Tools in Scientific Computing, pages 163–202. Birkha¨user Press, 1997.

Karypis, G. 2012. Parmetis - parallel graph partitioning and fill-reducing matrix ordering, 2012. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

Karypis, G. and K. Schloegel. 2011. *Parmetis: Parallel graph partitioning and sparse matrix ordering library*. Technical Report Version 4.0, Department of Computer Science, University of Minnesota, 2011.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2013	Final Report	

4. TITLE AND SUBTITLE

Development of Parallel GSSHA

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Paul Eller, Ruth Cheng, Aaron Byrd, Chuck Downer, and Nawa Pradhan

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
28FK8C

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Information Technology Laboratory
US Army Engineer Research and Development Center
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

8. PERFORMING ORGANIZATION REPORT NUMBER

ERDC TR-13-8

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Headquarters, US Army Corps of Engineers
Washington, DC 20314-1000

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

GSSHA (Gridded Surface Subsurface Hydrologic Analysis) is a physics-based, distributed, hydrologic, sediment and constituent fate and transport model. GSSHA can simulate 2D overland flow, 1D stream flow, 1D infiltration, 2D groundwater, and full coupling between groundwater, shallow soils, streams, and overland flow. GSSHA simulations can be very large and time consuming, simulating millions of grid cells over a period of years. In order to run these simulations in a reasonable amount of time, GSSHA must be parallelized to allow many processor systems to effectively run a single GSSHA simulation. Parallelizing GSSHA will enable shorter turnaround time, as well as the study of larger, more complex problems.

This work attempts to fully parallelize GSSHA using MPI, with the ultimate goal of being able to efficiently run GSSHA on thousands of processor cores. We hope to maintain the previous GSSHA functionality by allowing users to run GSSHA without MPI and/or without PETSc. At this point in time we have parallelized and tested the GSSHA code for the overland routing, infiltration, groundwater, soil erosion, channel routing, and lakes processes as well as many other routines needed to run these simulations correctly. We have also parallelized the secant Levenberg-Marquardt alternate run mode. For overland routing we have parallelized the ADE, ADE-PC, and explicit methods. For infiltration we have parallelized the Green and Ampt, Green and Ampt with Redistribution, Multi-layer Green and Ampt, and Richards methods.

15. SUBJECT TERMS		
Distributed	Sediment	Parallelization
Hydrologic	Gridded surface	GSSHA
	Transport model	Domain coupling

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED		31	